Algorithms & Data Structures Coursework Report

Krzysztof Czerwinski - 40283208

1 Introduction

In this coursework, we were to demonstrate our understanding of theory and practice of the contents of Algorithms and Data Structures module. We were asked to implement a text-based Tic-Tac-Toe game in C and write a report explaining our decision-making process when choosing appropriate data structures and algorithms to create our project.
I made a fully functioning application with the following major features:

- Text User Interface with user-friendly navigation
- customisation of game board and players
- recording and replaying games
- undo and redo functionality
- sophisticated AI using weights

2 Design

As always I stuck to consistent coding standards. I've chosen snake_case naming (UPPER_CASE for const/define) and _t suffix for types (structs) which isn't uncommon approach and may be found for example in GTK [1].
As C isn't an object-oriented programming language, I decided to treat structs as "classes", which makes the code a lot cleaner and easier to understand. All "methods" have prefixes indicating the struct they operate on, and their first argument is a pointer to a struct of this type (except factory functions and some menu functions). Functions intended to use outside (public) are declared in .h files and those needed only for the struct itself (private) directly in .c files. Given the size of my project, I'll briefly explain all files but focus only on the most essential parts and explain them in details.

main.c & game.h

That's the entry point of the game that loads settings, sets const variables, and displays menu.

menu

Initializes and prints Text User Interface using simple boxes and *submenu_t* data structures.

submenu

Holds two data structures used in menu. Submenu represents a single menu screen in the menu (ex. Main menu, Settings menu) and has pointers to the first and last option that it contains (it's a linked list).
option_t;
char text[32] - actual name of the option (ex. "Settings", "Exit").

char info[32] - additional information displayed at the bottom when this option is highlighted.
*void (*fun_ptr)(char*)* - pointer to the function that is executed when user chooses this option.
char args[32] - arguments that are passed to the fun_ptr function.
option_t prev* and *next* - pointer to the previous and next option.
submenu_t;
option_t first* and *last* - pointers to the first and last options in a submenu, together with *int count* (counter of how many options are in a submenu) makes up a linked list.
int pointer - number of the current highlighted option.
char title[32] - title of the submenu that is displayed at the top when a submenu is displayed.

functions

Contains functions that are used by **fun_ptr* in submenu options, all prefixed by *fun_*.

board

Contains all functions and *board_t* data structure needed to draw and play or replay a game.
board_t;
grid_t grid* - grid data structure with values of what pieces are in which cells.
int pointer_x and *pointer_y* - coordinates to the cell that is currently highlighted on the board.
int win_cond - number of pieces that must be in a row to end the game. That is set automatically in the *fun_play* (inside *functions.c*) to highest value between 3 and 5, no greater than width and height of a board (*Min(5, width, height)*).
player_t player_1* and *player_2* - pointer to the player data structures for both players that are playing on a board.
player_t current_player* - player that currently has their turn.
list_2_t undo* and *redo* - lists that contain coordinates of the previous and future moves respectively (used by undo/redo feature and game replaying). There is no need to save which Player placed which piece, it can be easily determined because players play alternately.
void board_draw(board_t*, bool);
Draws a board so it visually fits with the Text User Interface drawn by menu.
void board_play(board_t*);
Executes all logic needed to play a game on this board, including redrawing a board and additional information (ex. Whose turn is it), checking if game should end, executing player moves and calling AI functions (there are separate functions for some of those things).
void board_replay(board_t*);
A similar and much-shortened version of board_play function. It replays a game that is embedded in the *board_t* passed in the argument.
int board_check_win(board_t*);
Checks if there is a winner or a tie. It iterates through every cell and checks if there are enough pieces in a row, in four possible directions (right, down and two diagonals). If there is no winner yet, a linear search is performed on the board grid and if no empty cell is found, that means the board is full, and it's a tie.

player

Holds *player_t* data structure, players could be customized in the Settings menu. I decided to limit player names to 20 characters (+1 NULL terminator), that made it easy to load and saves names from replay files (program saves/loads the same amount of bytes each time).
player_t;
char name[21] - name of a player.
char piece - player's piece that is displayed on a board.
ai_t ai* - points to an AI controller, if it's NULL that means this is a human player.

vector

Vector data structure, *vector_2_t* holds 2 int values, *vector_t* one.
vector_2_t;
int x and *y* - vector values (*int value* in *vector_t*).
vector_2_t prev* and *next* - pointers to the previous and next vector.

list

Holds containers for vectors and functions to manipulate them.
list_2_t (and **list_t**);
vector_2_t first* and *last* - pointers to the first and last *vector_2_t* data structure in this list.
int count - number of vectors in the list.
vector_2_t* list_2_vector(list_2_t*, int);
Returns a pointer to the vector at particular index **int** from the list if index is less than *count*. Function moves through each vector in the list starting from *first* until iterator equals index and returns pointer.
void list_2_add(list_2_t*, vector_2_t*);
Adds a vector to a list. *Count* of the list increases, and pointers *prev*, *next* (inside vectors), *first* and *last* (inside list) are set to reflect changes.

grid

grid_t and *cell_t* data structures. Former is a linear data structure with immutable size that let us reference particular cell using two-dimensional coordinates.
grid_t;
cell_t cells* - pointer to cells. During construction, due to immutable nature of the grid continuous chunk of memory is allocated for all cells at once. We can later reference a particular cell using square brackets [].
int width and *height* - dimensions of a grid.
cell_t* grid_cell(grid_t*, int, int);
Returns pointer to a cell at particular x and y coordinates, or NULL if coords are outside of grid bounds.

ai

Contains data structure for representing an AI controller and all functions necessary to perform automated play against another player on board.
ai_t;

grid_t weights* and *weights_temp* are grid structures needed to store weights for calculations where to place next piece.
int pointer_x and *pointer_y* are set after performing weight calculations to the place where the next piece should be placed.
void ai_calculate(ai_t*);
That's the core function for determining the best move for an AI controller. It's based on the calculations that the board performs when checking if there's a winner and a simple weight-based approach. I've written the whole algorithm by myself, using my experience (past projects) and knowledge about AI.
It starts with setting all weight values in the ai controller to 0. Then it analyses the board searching for any places where there is a possibility of winning/losing. Calculations are performed in all eight possible directions (up, down, left, right and four diagonals). The more pieces in a row, the higher the chance of placing enough to end the game and AI will assign a higher weight to those cells. Where there's an opposing player on the way or not enough cells to place a sufficient amount of pieces in a row, cells will have much lower weight. After that AI found all possible patterns that could finish the game, but there's one more calculation performed. All eight neighbours of each cell are processed, and a weight is added to them to favour cells closer to already placed pieces and determine a balance between preventing to lose (blocking another player) and striving to win. It is possible to turn on weights displaying before each AI movement in Settings, to see results of those calculations in action. Finally, linear search is performed to find the best weight, and if there is more than one cell with the best value, a random one is chosen, and *pointer_x* and *pointer_y* values (of an *ai_t*) are set to point to this cell.
It doesn't make mistakes on purpose, but it's still possible to win because it doesn't learn or predict future moves, it just calculates the best move given the current state of the board. Because of the random element (choosing random cell if the best weights are the same in more than one), it also doesn't play perfectly. A good example of that is to play AI vs AI games (with undo/redo disabled - the game will last a few seconds), there is sometimes a winner.

settings

Functions for managing game settings: creating and destroying settings struct, saving and loading settings from files. It uses binary writing and reading for convenience. Settings file is created automatically if it isn't found when the game starts.
settings_t;
Struct holds all data that is saved and loaded to a file to preserve game settings across sessions.
char* settings_new_replay(settings_t*);
Function returns next available name for the replay file by checking if file with a particular name already exists, starting from 0.replay, then 1.replay.

file

Contains utility functions for managing files and handles saving and loading of replays.
bool replay_save(board_t*);
Saves board state to a file so it can be loaded and replayed. All data is saved, so we can replay a game the way it exactly was played (players name, pieces, etc. are saved).

board_t* replay_load(**char***);
Creates a board, loads all data from a file and returns a pointer to it, so it's ready to be replayed.
list_t* replay_list(**int**);
Returns a list of vectors with replays available in the working directory.
const char* get_file_ext(**char***);
Returns a file extension.
bool file_exists(**char***);
Returns true if there is a file with a particular name in the working directory.

test

Two functions used to test execution time anywhere in code.

3 Enhancements

Many things could be changed or improved, but it wasn't always possible. One of the more significant problems was that it's impossible to implement multithreading in C (on Windows) without using external libraries. That resulted in many consequences, one of which is that the program always stops when it waits for user input. Therefore, once automatic replaying is turned on, it can't be terminated without exiting the game. I was initially thinking about implementing a time limit for a player move, but again that isn't doable without at least two threads. Without Windows API it's also impossible to change colours - the whole game is monochromatic. Freely moving the carriage on the screen is also impossible in pure C. Therefore the entire screen must be redrawn each step (that results in jittering, especially when there are many changes in a short period). It's impossible to list files in a folder, so I needed to use a trick when searching for replay files (replays are just consecutive numbers, and program checks one file at a time). Saving and loading a game using replay files wouldn't be very complicated to implement. I was thinking about implementing a random-based generation for AI names using predefined words, ex. Good Player, Bad Gamer, Almighty Being. This could yield interesting and funny names.
I'll list the rest briefly, as those are self-explanatory: turning recording on and off, removing replays in-game, turn count during gameplay, determining that win is impossible after each turn, timestamp in replays.

4 Critical Evaluation

Initially, I was thinking about implementing command-base navigation as an alternative to the current, interactive approach but I realised that is unnecessary. That would also require letters and digits on the border of a board, so we could reference each cell easily.
Text User Interface could be vastly improved using box-drawing characters, but it would require a much complex level of abstraction (due to corners/midsections in borders) to the current menu system which I find unnecessary for this project.
I was also thinking about making board pointer move only to empty cells, but it works just fine the way it is now, and that would require another complex algorithm.
Implementing linear data structures, so the main container contained a pointer to the first and last element, and a count was a good solution. That made it possible for an easy linear search from the first element and adding a new item at the end.

I considered using stacks for undo/redo features. That would include more code to already implemented list type, which is completely unnecessary. I was initially using *list_2_t* even if I needed single-value vectors. Later I implemented *list_t* and *vector_t* for those cases, but improvement in execution time was negligible. A conclusion might be drawn that this change was unnecessary, but although execution time didn't improve, a lot of memory was saved.
It was hard for me to find anything else worth improving so the execution time would improve. I was trying to choose fast solutions from the beginning. I could change *cell_t* (an element of *grid_t* which contains just one int) to an int, but the creation time for the biggest *board_t* and *ai_t* (both of those structures contain *grid_t*) is already displayed as precisely 0, so there is small to none possible improvement.

5 Personal Evaluation

I think this coursework was a big challenge for me, but I enjoyed it. I wouldn't say I like C, but I got used to it after a few days of work and learned a lot. Later on, it was a pleasure to work on this project; it was much better than I expected.
I fell that I spent too much time abstracting things, but I found that necessary due to the size of my project. I still have a feeling that the general structure isn't the way it should be, but that's probably due to my inexperience with C.
Makefile was a challenge, it was hard to find a good explanation of makefile for nmake, but I finally found an excellent example of an automated one [2].
I used *System("cls");* to clear the console. I'm not too fond of this solution, it calls an external program, and therefore it's slow and a bit unreliable (I can't be sure if it'll work on every Windows), but there was no other way without external libraries/windows API.
I might spend too much time working on AI algorithms (it was rewritten three times/20+ hours), but I'm satisfied with the results, so I don't think that was a waste of time.
I'd also like to point out that I didn't use any sorting algorithm; there just wasn't any necessity for it. There is a lot of various data structures, and I needed some basic search algorithms (linear search).
I was searching for solutions to my problems on the internet, I didn't keep track of all websites I used; but mainly those were: Stack Overflow [3], GitHub [4], Tutorialspoint [5], GeeksforGeeks [6].

6 References

1. GNOME GTK - https://github.com/GNOME/gtk
2. Makefile https://gist.github.com/serialhex/ada2b37581591716d41f70edd2986938
3. Stack Overflow - https://stackoverflow.com/
4. GitHub - https://github.com/
5. Tutorialspoint - https://www.tutorialspoint.com
6. GeeksforGeeks - https://www.geeksforgeeks.org